Turing's Ideas and Models of Computation

Eugene Eberbach[1], Dina Goldin[2], and Peter Wegner[3]

[1] Computer and Information Science Department, University of Massachusetts, North Dartmouth, MA 02747, eeberbach@umassd.edu
[2] Computer Science & Engineering Department, University of Connecticut, Storrs, CT 06269, dqg@cse.uconn.edu
[3] Department of Computer Science, Brown University, Providence, RI 02912, pw@cs.brown.edu

Summary. The theory of computation that we have inherited from the 1960's focuses on algorithmic computation as embodied in the Turing Machine to the exclusion of other types of computation that Turing had considered. In this chapter we present new models of computation, inspired by Turing's ideas, that are more appropriate for today's interactive, networked, and embedded computing systems. These models represent *super-Turing* computation, going beyond Turing Machines and algorithms. We identify three principles underlying super-Turing computation (*interaction with the world*, *infinity of resources*, and *evolution of system*) and apply these principles in our discussion of the implications of super-Turing computation for the future of computer science.

1 Introduction: Algorithmic computation

Alan Turing is known mostly as the inventor of *Turing Machines*, which he created in an effort to formalize the notion of *algorithms*.

> **Algorithm:** systematic procedure that produces – in a finite number of steps – the answer to a question or the solution to a problem [2].

This notion well precedes computer science, having been a concern of mathematicians for centuries. It can be dated back to a 9-th century treatise by a Muslim mathematician Al-Koarizmi, after whom algorithms were named. *Algorithmic computation* refers to the computation of algorithms.

> **Algorithmic computation:** the computation is performed in a *closed-box* fashion, transforming a finite input, determined by the start of the computation, to a finite output, available at the end of the computation, in a finite amount of time.

Turing Machines have the following properties that model algorithmic computation:

- their computation is *closed* (shutting out the world);
- their resources (time and memory) are *finite*;
- their behavior is *fixed* (all computations start in same configuration).

The Turing Machine model formed the foundations of current theoretical computer science. This is largely due to the *strong Turing Thesis*, found in popular undergraduate textbooks, which equates Turing Machines with *all* forms of computation:

> **Strong Turing Thesis**: A Turing Machine can do everything a real computer can do [35].

It is little known that Turing had proposed other, non-algorithmic models of computation, and would have disagreed with the strong Turing Thesis. He did not regard the Turing Machine model as encompassing all others.

As with many of his other ideas, Turing was far ahead of his time. Only now, with the development of new powerful applications, it is becoming evident to the wider computer science community that algorithms and Turing Machines do not provide a complete model for computational problem solving.

Overview. We start with a discussion of Turing's rich contributions to computer science, focusing on various models of computation. Next, we present examples of super-Turing computation, which is more powerful than Turing Machines, and discuss the three principles that contradict the algorithmic properties above, making it possible to derive super-Turing models:

- interaction with the world;
- infinity of resources;
- evolution of the system.

While the strong Turing thesis denies the existence of *super-Turing* models, we explain why Turing was not the author of this thesis and would disagree with it. We show that super-Turing models are a natural continuation of Turing's own ideas. We then discuss how super-Turing computation might influence computer architecture, programming paradigms and the foundations of computer science. We conclude on a philosophical note.

2 Turing's contributions to computer science

2.1 The *Entscheidungsproblem* and Turing's Automatic Machines

Alan Turing was born in 1912, and enrolled at King's College in Cambridge in 1931 as a mathematics undergraduate. He became a fellow of King's College in 1934, completing a dissertation on the Central Limit Theorem. He then became interested in the *Entscheidungsproblem* (decision problem), one of the most alluring conjectures in mathematics, proposed by the prominent mathematician David Hilbert in 1918.

Hilbert's conjecture that any mathematical proposition could be *decided* (proved true or false) by mechanistic logical methods was unexpectedly disproved by Gödel in 1931, who showed that for any formal theory, there will always be undecidable theorems outside of its reach. Mathematicians like Alonzo Church [5] and Turing continued Gödel's work, looking for alternate techniques for proving this undecidability result.

Turing's proof, provided in his 1936 paper [37], "On Computable Numbers with an Application to the Entscheidungsproblem" was based on a novel model of *automatic machines* (a-machines), which can be built to carry out any algorithmic computation. Turing showed that despite their versatility, these machines cannot compute all functions; in particular, he proved that the now-famous *halting problem* is undecidable.

The a-machine model consists of:

- a one-dimensional *erasable tape* of infinite length, capable of storing symbols, one on each cell of the tape;
- a read/write *tape head* capable of moving to the left or right on the tape, and of retrieving or storing one tape symbol at a time at the current tape location;
- a *control mechanism* that may be in any of a bounded number of states;
- a *transition table* which, given the symbol under the tape head and the current state of the machine, specifies the next action of the tape head and the new state of the machine.

At the beginning of a computation, the machine is in a special *initial* state. At each step of the computation, the machine's control mechanism causes one symbol to be read from the current tape location. The control mechanism then looks up in the transition table actions that depend on the value of the retrieved symbol as well as on the current state of the machine. It then writes a new symbol at the current tape location, transitions to a new state, and moves left or right one cell on the tape. The computation terminates once the machine reaches a special *halting* state.

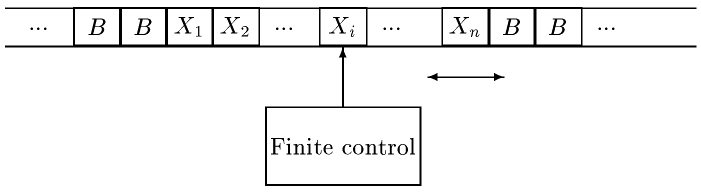

Fig. 1. Turing Machine

We now know a-machines as *Turing Machines*, and this is how we will refer to them for the rest of this chapter. Turing Machine computations have the following properties:

- The TM models *closed* computation, which requires that all inputs are given in advance;
- The TM is allowed to use an unbounded but only *finite amount of time and memory* for its computation;
- Every TM computation starts in an identical *initial configuration*; for a given input, TM behavior is fixed and does not depend on time.

These properties are perfectly suitable for modeling *algorithmic* computation. Note however that they prevent TMs from modeling directly many aspects of modern computing systems, as we discuss in section 3.

Turing Machines were adopted in the 1960's, years after Turing's premature death, as a complete model for algorithms and computational problem solving. They continue to serve as a standard model of computation to this day. Turing's 1936 paper has come to represent the birth of computer science, though the motivation and substance of this paper were entirely mathematical.

2.2 Algorithms and the history of the Turing thesis

In the early 1930's, the quest to prove (or disprove) Hilbert's famous *Entscheidungsproblem* led to many new classes of functions. Gödel defined *recursive functions*. Soon thereafter, Church invented λ-*calculus*) and showed that it defines the same class of functions [5]. A third class of functions, those computable by Turing Machines (TMs), was established by Turing around the same time, and also proved equivalent to recursive functions([37]).

Both Church and Turing were in search of *effective* ways of computing functions, where "effectiveness" was a mathematical notion synonymous with "mechanical" and lacking a formal definition. Church proposed to identify the notion of an effectively calculable function with the notion of a λ-definable function. Turing made the same appeal on behalf of his machines [37]:

> **Turing's thesis**: *Whenever there is an effective method for obtaining the values of a mathematical function, the function can be computed by a Turing Machine.*

Very soon thereafter, Turing established that Church's lambda-definability and his own machine-based computability are equivalent [37].

Note that the infinite length of the TM tape plays a key role in this result. With a bound on the length of the tape, only finitely many different configurations of the tape are possible, reducing the TM to a *finite-state automaton* (FSA). The class of FSAs, corresponding to *regular languages*, is known to be less expressive than the class of TMs. The infinity of TM tape is reconciled with the finiteness of the physical world by recognizing that it is a formal model rather than an actual machine. Whereas a physical machine can only have a finite tape, this tape can be upgraded to a longer one when needed; hence a proper model for it is one where the tape is infinite.

The equivalence of the three classes of functions (λ-definable, recursive, and Turing-computable) was taken as confirmation that the notion of effective function computation had finally been formally captured. The claims of Church and Turing are usually combined into one, known as the Church-Turing thesis:

> **Church-Turing thesis**: *The formal notions of recursiveness, λ-definability, and Turing-computability equivalently capture the intuitive notion of effective computability of functions over integers.*

However, Gödel much preferred Turing's approach, since its identification with effectiveness is more immediate [8].

While this work was purely mathematical, and applied only to *functions over integers*, it had a strong influence on the field of computer science when it emerged as a mature discipline decades later. In particular, the robustness of the notion of effective function computation has served to give it a central role in the foundation of computer science. Turing Machines provided this new field with legitimacy on par with physics and mathematics, by establishing it as the study of a class of concrete and well-defined phenomena:

> **Physics:** study of properties of matter;
> **Mathematics:** study of quantity and space;
> **Computer Science:** study of algorithmic problem solving computable by Turing Machines.

While the Church-Turing thesis only applied to the effectiveness of computing functions over integers, it extends easily to functions over finite strings, since strings can be naturally encoded as integers. However, it does not extend to other types of computation, such as functions over real-valued inputs, or such as interactive computation. Both Church and Turing were aware of this limitation. Whereas Church's concerns did not go beyond functions over integers, Turing's concerns were more general. As we discuss next, he also proposed machines for interactive computation, distinct from a Turing Machine.

2.3 Turing's quest for interaction: Choice Machines and Oracle Machines

Automatic machines (a-machines) were not the only model introduced by Turing in his 1936 paper. In the same paper, Turing also proposed *choice machines* (c-machines) as an alternate model of computation. Whereas a-machines operate in a closed-box fashion as if on "automatic pilot" (hence their name), c-machines interact with an operator such as a human user during the computation. In Turing's words, a c-machine's "motion is only partially determined by the configuration"; in certain configurations, it stops and "cannot go on until some arbitrary choice has been made by an external operator" [37].

Choice machines were introduced by Turing as an alternative conceptualization of computation, one that is interactive. Turing clearly realized that the algorithmic computation of automatic machines is not the only type of computation possible. However, Turing's goal in [37] was to prove the unsolvability of the *Entscheidungsproblem* rather than to set standards for models of computation. Formalization of a-machines enabled him to reach his goal, so the bulk of the paper was concerned with them rather than with c-machines.

Eventually, a-machines were adopted as the standard model of computation, and c-machines Some believe that *oracle machines*, introduced by Turing just a few years later [38], provide a formalization of *c-machines*, making them unnecessary.

There is indeed an important similarity between choice machines and oracles machines: both make queries to an external agent during the computation. In the case of an oracle machine, this agent is an *oracle* rather than a human operator. This oracle is formally described as a set that can be *queried* about any *value*; it returns *true* if the queried value is in this set and *false* otherwise.

In ancient Greece, *oracles* were people whom others consulted for advice; they were believed to possess access to hidden knowledge that came to them directly from the deities. Just as Greek oracles have super-human knowledge, Turing's oracles are meant to represent *uncomputable* information obtained from outside the system. Turing specifically excluded the possibility that the oracle was an effective computing entity:

> *We shall not go any further into the nature of this oracle apart from saying that it cannot be a machine* [38].

Since oracles cannot be machines, do they model humans, such as the operators of c-machines? The set-based semantics of oracles preclude this possibility. Because they are sets, oracles are *static*, and the outcome of each query is predetermined before the oracle machine (*o-machine*) starts computing. During the computation, the same query will always yield the same answer from a given oracle. Clearly, the same cannot be said of humans, for whom the same question can yield a different answer depending on their mood or some other circumstance. Hence, Turing's *choice machines* are not just *oracle machines* by another name, but a different model of computation.

2.4 Turing's contribution to cryptology and complexity theory: work on Enigma and Colossus

During War World II, Alan Turing worked as a top cryptanalyst and chief consultant at the Government Code and Cipher School at Blechley Park. By using his previous experience with ciphers, as well as his knowledge of combinatorics, code theory and statistics, Turing contributed substantially to breaking the code of the *Enigma*, the encrypting/decrypting machine that the German navy used for all its radio communications. More importantly, he mechanized the decryption process, using an electromechanical machine of his invention – the *Turing Bombe*. Later, a much faster device – the *Colossus* – was deployed; it can be considered the world's first electronic computer.

Just as the primary purpose of Turing Machines was to show that there exist problems which cannot be solved by mechanical (algorithmic) means, Turing's work at Blechley Park dealt with finding the mechanical means to solve problems that are hard, but not impossible. The art of breaking codes required the ability to deal with the intricate complexity of multi-level encryption algorithms. This was Turing's direct practical contribution to cryptography and indirectly to complexity theory. In fact, he pioneered what now would be called an interactive randomized approach to breaking ciphers, by intelligent guessing the key based on the statistical evidence, and exploring the loopholes in the Germans' use of Enigma. This, together with the speed of the Bombe and then Colossus, sliced through the intractable search space of the problem.

Turing was also chief liaison between American and British cryptanalysts. The work of British scientists at Blechley Park was highly regarded by British prime minister Winston Churchill, and recognized as one of the deciding factors in winning the war. However that work, because of its top secret nature, remained unpublished and unknown for many years.

2.5 Turing's work on ACE as a precursor of general purpose universal computers

After the end of the war Turing joined the National Physical Laboratory in 1945 to work on the *Automatic Computing Engine (ACE)*. ACE was one of several postwar attempts to build a working computer; other contemporary projects include EDVAC and IAS computers in the USA (at UPenn and Princeton, respectively), and Wilkes EDSAC machines at Cambridge in the UK.

Turing's plan was to build the first programmable general-purpose computer, which would keep both the code and the data in memory, and execute the code over the data. Turing drew his inspiration for ACE directly from his earlier theoretical work. As he said in a lecture to the London Mathematical society [40],

> Machines such as the ACE may be regarded as practical versions of [the Turing Machine]. There is at least a very close analogy.

In particular, Turing saw a direct parallel between the capability of ACE to accept and execute programs, and his notion of a *universal Turing Machine*, which was part of his 1936 *Entscheidungsproblem* paper [40]:

> the complexity of the machine to be imitated is concentrated in the tape [of the Universal Turing Machine] and does not appear in the

> universal machine proper... This feature is paralleled in digital computing machines such as the ACE. They are in fact practical versions of the universal machine... When any particular problem has to be handled the appropriate instructions... are stored in the memory of the ACE and it is then 'set up' for carrying out that process.

Turing's vision of a programmable computer is in stark contrast to all other computer designs of that time, including the Colossus and the ENIAC, which required a manual reconfiguration for every new computation. It is not surprising that ACE's computer architecture was very different from, and more complicated than, other computer designs of the time.

ACE was planned to be the fastest (1 microsec. clock cycle) and having the largest memory in the world (60,000 bits). ACE's design [39] involved many pioneering concepts that become standard part of computer architecture decades later. ACE would have a large set of 32 general-purpose registers, and a simple hardware system implementing fast basic minimum of arithmetical and Boolean functions; Turing believed that other functions (including floating-point arithmetic) should be programmed. This pioneering approach to hardware design did not receive due recognition until the early 1980's, when it became known as *RISC* (Reduced Instruction Set Computers).

Turing also pioneered *subroutine hierarchies* (then called *instruction tables*) which can be viewed as a precursor to high-level programming languages. Turing invented calling stacks for invoking them (then called *bury* and *unbury* routines). He proposed self-modifiable code to implement conditional branching. By contrast, the competing designs of the EDVAC, the IAS, and the EDSAC machines were based on *accumulator* architecture, whose design was relatively straightforward.

Turing's vision to build a machine that would show "genuine intelligence" went far beyond the project's original mission of doing "large difficult sums" and frightened his superiors. The administration of the National Physical Laboratories, under the directorship of Sir Charles Darwin (grandson of the well-known English naturalist) found Turing's ideas too revolutionary, especially in the context of postwar British frugality.

In retrospect, the ideas embodied in the design of ACE make perfect sense. In time, many aspects of Turing's ACE design have proved as valuable as the Turing Machine and the Turing test (section 2.7). But without the benefit of hindsight, Turing's contemporaries working on similar projects, such as Maurice Wilkes, were skeptical whether a machine of such complexity and with such revolutionary architecture would ever work. The ACE project never received the funding it sought, and the disappointed Turing left the National Physical Laboratory in 1948 for the University of Manchester.

2.6 Turing's Unorganized Machines as a precursor of neural networks, evolutionary computation and reinforcement learning

Before leaving for Manchester in 1948, Turing produced a final report on ACE which can also be viewed as a blueprint for the future field of *neural networks.* Titled *Intelligent Machinery* [41], this report was left unpublished until 1968, because Darwin considered it to be a "schoolboy essay" not suitable for publication.

In this report, among other futuristic ideas, including robots taking country walks, Turing proposed new models of computation, which he called *unorganized machines (u-machines).* There were two types of u-machines, those based on *Boolean networks* and those based on *finite state machines* [41,36]. Turing took his inspiration from the working of the human cortex, and its ability for self-adaptation.

- **A-type** and **B-type** u-machines were Boolean networks made up of two-input NAND gates (*neurons*) and synchronized by global clock; the number of neurons remained fixed. While in A-type u-machines the connections between neurons were fixed, B-type u-machines had modifiable switch type interconnections. Starting from the initial random configuration and applying a kind of genetic algorithm, B-type u-machines were supposed to learn which of their connections should be on and which off.
- **P-type** u-machines were tapeless Turing Machines reduced to their Finite State Machine control, with an incomplete transition table, and two input lines for interaction: the *pleasure* and the *pain* signals. For configurations with missing transitions, the tentative transition to another state could be reinforced by pleasure input from the environment, or cancelled in the presence of pain.

In his B-type u-machines, Turing pioneered two areas at the same time: *neural networks* and *evolutionary computation*; his P-type u-machines represent *reinforcement learning.* However, this work had no impact on these fields, due to the unfortunate combination of Turing's death and the twenty-year delay in publication. As a result, others got the credit for these ideas:

- *Neural networks* are typically credited to Pitts and McCulloch neurons (1943) [21] and Rosenblatt's perceptron (1958) [30].
- *Evolutionary computation* is typically credited to Holland's work (1968) [1] on genetic algorithms, although it is acknowledged that the area has been rediscovered around ten times between the 1950's and 1960's [14].
- *Reinforcement learning* has been been attributed to Minsky and Farley and Clark papers from 1954 [26,13], or Samuel's checkers program from 1959 [32].

Turing was convinced that his B-type u-machine can simulate his Universal Turing Machine, though he never provided a formal proof. In order to simulate the infinite tape of a Turing Machine, a u-machine with an infinite number of neurons would be needed. This is due to the *discrete* nature of the neurons, which were based on two input Boolean NAND gates. By contrast, two *real-valued* neurons are sufficient to model a Turing Machine.

B-type u-machines were defined to have a finite number of neurons, and it is not clear whether Turing was aware that infinitely many neurons were needed for the simulation. This inconsistency would certainly have been uncovered when working on the formal proof. But perhaps Turing was aware of it, and expected to have no problems extending his definitions to the infinite case.

2.7 Turing as a founder of Artificial Intelligence

In this section, we explore Turing's contributions to *Artificial Intelligence* (AI), of which he is considered one of the founders. Turing's interest in AI can be traced at least to his final report on ACE, where he envisions "intelligent" behaviors of future generations of computers [41]. At that point, intelligence was viewed mainly in terms of a *search strategy*; an intelligent agent is one that can find the best action based on current knowledge.

Turing identified chess as a good starting point for exploring intelligent search strategies. Ever optimistic about the progress of computing research, Turing estimated in 1941 that computers would be able to beat human chess champions by about 1957. To make progress towards this goal, Turing and David Champernowne wrote the *Turochamp* chess program in 1948, applying a search strategy known as *Minimax* towards choosing the next move. Eventually, computers were built that could beat human chess champions, but it took 40 years longer than Turing predicted. Using a variant of Minimax known as the alpha-beta search, a supercomputer named *Deep Blue* beat the world chess champion Garry Kasparov in 1997.

Turing expected computers to match humans not only in chess, but in every intelligent endeavor. In a famous 1950 paper [42] Turing provocatively led with the question "Can Machines Think?" and proposed his famous test for intelligence. Now known as the *Turing Test*, it is based on the "imitation principle":

> **Turing Test for AI**: *If a computer, on the basis of its written responses to questions, could not be distinguished from a human respondent, then one has to say that the computer is thinking and must be intelligent.*

The imitation principle, inspired by the then-popular behaviorist school of thought, stated that there was no way to tell that other people were "thinking" or "conscious" except by the process of comparison with oneself. Thus if computers behaved like people (i.e., the "black-box" approach, independently how they are internally built), they should be credited with human-like attributes of consciousness, intelligence, or emotion. As the result, a new field of *artificial intelligence* grew up around the problems of defining and duplicating consciousness and intelligence.

The Turing test, and the early artificial intelligence research which was based on it, attracted much criticism. In order to explore the arguments of the critics, it is useful to break down the question "Can Machines Think" into two separate questions:

> **(Extensionality)** *Can machines simulate the behavior associated with thinking?*
> **(Intensionality)** *Can we say that machines that simulate thinking are actually thinking?*

Turing answered "yes" to both of these questions, while the critics were divided into those who believed that machines cannot simulate thinking (*extensional skeptics*) and those who believed that simulation without understanding does not capture the essence of thinking (*intentional skeptics*). Extensional skeptics place limits on the expressive power of computation, while intensional skeptics reject the behaviorist view that unobservable inner attributes are irrelevant.

The second question has a metaphysical flavor, and is therefore outside the scope of a computer science inquiry. However, the first question is very much of interest to computer scientists, especially when rephrased as: *"Can machines act intelligently?"* We explore this question in the context of interactive models of computation in section 5.1.

2.8 Turing as a precursor of Artificial Life

In 1952, Turing published a paper on morphogenetic theory in the Royal Society Proceedings [43]. Turing tried to capture the growth and pattern occurrences in plants and animals, describing them as dynamical systems, in the form of nonlinear differential equations. Turing was fascinated by the appearance in nature of the Fibonacci numbers series in the leaf arrangements and flower patterns.

This work, along with that of John von Neumann, can be considered as precursors of there area of *artificial life* and of *biogenetic computing.* Von Neumann's work considered the universal computability and universal constructibility of cellular automata, based on his theory of self-reproducing automata from 1952.

Both Von Neumann and Turing's work on artificial life remained unfinished because of the authors' death. However, Arthur Burks edited and published von Neumann manuscript in 1966 [45], while Turing's work in this area remained practically unknown. As a result, usually only von Neumann is given full credit for founding the area of artificial life.

3 Super-Turing Computation

In this section we discuss why Turing Machines (TMs) do not represent a complete theory for problem solving. In particular, the three properties of TM computation, while perfectly suited for modeling algorithmic computation, act to prevent it from modeling directly many aspects of modern computing systems:

- TM computations are *closed*, which requires that all inputs are given in advance;
- TM computations are allowed to use an unbounded but only *finite amount of time and memory*;
- TM computations all start in an identical *initial configuration*; for a given input, TM behavior is fixed and does not depend on time.

By contrast, modern computing systems process *infinite streams* of *dynamically generated* input requests. They are expected to continue computing *indefinitely* without halting. Finally, their behavior is *history-dependent*, with the output determined both by the current input and the system's computation history.

A large percentage of the computer-science community believes that while Turing Machines are not very convenient to model some aspects of computation, they nevertheless cover all possible types of computation. As a consequence, it is considered futile to look for models of computation going beyond TMs. In this section, we identify the source of this common misconception, and discuss three different directions for extending TM computation to super-Turing computation: *interaction*, *infinity*, and *evolution*.

These extensions can be contrasted with the many failed attempts to break out of the "Turing tarpit" of TM-equivalent computation known to us from theory of computation textbooks, such as increasing the number of Turing Machine tapes. Those attempts always remained in the algorithmic paradigm, and consequently were doomed to fall within the bounds of the Church-Turing thesis. By contrast, each of these three extensions lifts computation out of the algorithmic paradigm, by redefining the space of computational problems. What is being computed is no longer just fixed functions from integers to integers (or some equivalent), but also non-algorithmic computational problems, or tasks.

3.1 The strong interpretation of the Turing thesis

A *Universal Turing Machine* is a special Turing Machine introduced by Turing in [37], that can simulate any other Turing Machine – hence its name. It served as the inspiration for the notion of general-purpose computing (section 2.5).

The principle of universality can easily be extended to any other class of machines for computing functions. As long as each machine in this class can be captured by a finite description which defines what this machine "would do in every configuration in which it might find itself" [40], a Turing Machine can be created to simulate all machines in this class:

> **Universality thesis**: *Any class of effective devices for computing functions can be simulated by a Turing Machine.*

Both the Turing thesis (section 2.2) and the Universality thesis constitute fundamental yet distinct contributions to the theory of computation, which was established as a separate area of computer science in the 1950's. It is astonishing that these results were accomplished by Turing simultaneously in his seminal paper on Turing machines [37], which predated any computers.

The original digital computers were in fact devices for computing functions, much like Turing Machines; their architecture did not allow any interaction during the computation. This led to the equating of computers with algorithmic computation, and to the following (incorrect) corollary of the universality thesis:

> **Universality corollary**: *Any computer can be simulated by a Turing Machine.*

While the universality corollary is true when computers are limited to the task of computing functions or algorithms, it does not apply in the context of today's highly interactive computing systems such as the Internet.

When the first undergraduate computer science textbooks were being written in the 1960's, Turing's contributions to theory of computation needed to be presented in a more accessible fashion. As a result, the Turing thesis and the Universality corollary were glibly combined into one, resulting in the following (incorrect) *strong interpretation* of the Turing thesis that one sees in undergraduate textbooks:

> **Strong Turing thesis**: *Whenever there is an effective method for solving a problem with a computer, it can be computed by a Turing Machine.*

The current generation of computer scientists has absorbed the strong interpretation of the Turing thesis with their undergraduate education, believing it a heresy to question it. However, this interpretation needs to be distinguished from the actual contributions of Turing or his contemporaries on which it is based. There is no question that both the Turing thesis and the Universality thesis only applied in the realm of functions over integers. When these sources of the strong Turing thesis are reexamined, its position as a dogma of computer science becomes shaky. The strong Turing thesis needs to be recognized as incorrect – despite common belief to the contrary.

3.2 Driving Home From Work

In this section, we discuss the problem of *driving home from work* (DHW) [46], which cannot be solved algorithmically, but is nevertheless computable. The existence of computable problems that a Turing Machine cannot solve contradicts the strong Turing Thesis, proving it incorrect.

The *DHW* problem. Consider an *automatic car* whose task is to drive us across town from work to home. The output for this problem should be a time-series plot of signals to the car's controls that enable it to perform this task autonomously. How can we compute this output?

In the *algorithmic* scenario, where all inputs are provided *a priori* of computation, the input to the *DHW* problem includes a map of the city which must be precise enough to compute the exact path the car will take. This scenario, typical of AI approaches to similar problems through most of the second half of the last century, is illustrated in Figure 2.

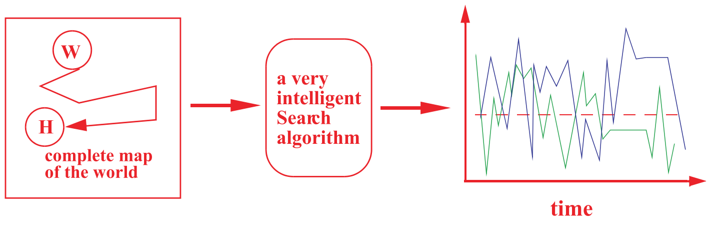
W
H
complete map of the world
a very intelligent Search algorithm
time

Fig. 2. Driving home from work: the algorithmic scenario

Note that in addition to the map, the input needs to specify the exact road conditions along the way, including every pothole and every grain of sand. By the principles of *chaotic behavior*, such elements can greatly affect the car's eventual course – like the Japanese butterfly that causes a tsunami on the other end of the world.

In a *static* world, this input is in principle specifiable, but the real world is *dynamic*. The presence of mutable physical elements such as the wind and the rain affect the car's course, both directly (as the wind blows at the car) and indirectly (as the wind shifts the sand in the path of the car). It is doubtful whether these elements can be precomputed to an accuracy required for the *DHW* problem.

We can remain optimistic until we remember that the world also includes humans, as pedestrians or drivers. To avoid collisions, we must precompute the exact motions of everyone who might come across our way. To assume that human actions can be computed ahead of time is tantamount to an assertion of *fatalism* – a doctrine that events are fixed in advance so that human beings are powerless to change them – clearly beyond the purview of any computer scientist. Therefore, we must conclude that the *DHW* problem is unsolvable:

Computational tasks situated in the real world which includes human agents are not solvable algorithmically.

Nevertheless, the *DHW* problem *is* computable – interactively, with a *driving agent*. In this scenario, the agent's inputs, or *percepts* [31], consist of a stream of images produced by a video camera mounted on the moving car. The signals to the car's controls are generated by the agent *on-line* in response to these images, to avoid steering off the road or running into obstacles.

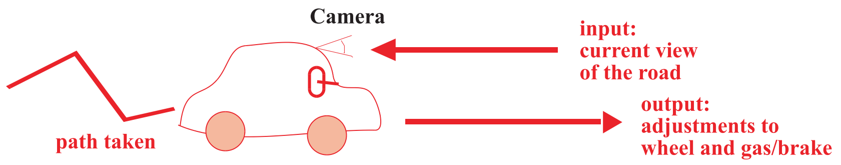

Fig. 3. Driving home from work: the interactive scenario

This change in the scenario, illustrated in figure 3, is akin to taking the blindfolds off the car's driver, who was driving from memory and bound to a precomputed sequence of actions. Now, he is aware of his environment and uses it to guide his steering.

The DHW example proves that there exist problems that cannot be solved algorithmically, but are nevertheless computable.

Note that we have not just restructured the inputs, but also changed the *model of computation* as well as the notion of a *computational problem*. Algorithmic problems are computed *off-line*; the output is generated *before* the driving begins. Interactive problems are computed *on-line*; the output is generated as the car drives. Furthermore, the inputs and outputs for interactive computation are interdependent; decoupling them, such as replacing the videocamera with a prerecorded videotape of the road, will be tantamount to putting the blindfolds back on the driver.

3.3 Super-Turing computation

We refer to computation that violates the strong interpretation of the Turing thesis (section 3.1) as *super-Turing computation*; *driving home from work* was an example.

Super-Turing computation: *all computation, including that which cannot be carried out by a Turing Machine.*

Super-Turing computation is more *powerful* than the algorithmic computation of Turing Machines, in that it can solve a wider class of computational problems. Our use of the term *super-Turing* is meant to have a positive connotation. We do not consider the higher expressiveness of new computing models as something excessive, but rather as a desirable feature, and as a natural continuation of Turing's ideas.

We identify three principles that allow us to derive models of computation more expressive than Turing Machines:

- interaction with the world;
- infinity of resources;
- evolution of the system.

We discuss these principles next.

Interaction with the environment. In his 1936 and 1939 papers, Turing showed that Turing Machines were only appropriate for computing recursive functions over integers, and proposed *choice machines* (*c*-machines) and *oracle machines* (*o*-machines) as alternate models that supported richer forms of computation than Turing Machines [37,38]. Both of these models extend Turing Machines by *interaction.*

> *Interactive computation* involves interaction with an external world, or the *environment* of the computation, *during* the computation – rather than *before* and *after* it, as in algorithmic computation.

Driving home from work (section 3.2) is an example of interactive computation. As another example, consider missile trajectory computations. This was an early application of computers, dating to World War II. In the original (algorithmic) scenario, the input includes the location of the target and the flight characteristics of the missile; the output is the direction and angle at which to launch the missile so it (hopefully) hits the target. By contrast, the computation for today's smart missiles is interactive. Once launched, they continue to monitor their progress and to adjust their trajectory to remain on target. In the presence of wind gusts and air pressure changes, interaction has proven necessary to assure accurate long-distance targeting.

Two types of interactive computation can be identified [47]. *Sequential interaction* describes the computation of a single agent, such as the smart missile, as it interacts with its *environment.* All inputs are interleaved into a single *input stream*, and are processed sequentially. By contrast, *distributed interaction* involves many agents working concurrently in an asynchronous fashion, with multiple autonomous communication channels which can be automatically reconfigured during the computation.

While algorithmic problems are solved by algorithmic systems such as Turing Machines, interactive problems are those solved by interactive systems:

> **Algorithmic problem**: transforming input strings to output strings
> **Interactive problem**: carrying out a computational task or service

The intuition that computing corresponds to formal computability by Turing machines breaks down when the notion of what is computable is broadened to include interaction. Though the Church-Turing thesis is valid in the narrow sense that Turing Machines express the behavior of algorithms, the broader assertion that algorithms precisely capture what can be computed is invalid [47]. By interacting with the external world, interactive systems can solve a larger class of problems, such as driving home from work or the smart missile.

Interactive computation has been captured under many different forms, such as *concurrent*, *distributed*, or *reactive* computation. It is a different computational paradigm, which expands the notion of a computational problem [46,49]. The paradigm shift from algorithms to interaction captures the technology shift from mainframes to workstations and networks, from number crunching to embedded systems and user interfaces, and from procedure-oriented to object-based and distributed programming.

Greater problem-solving power is synonymous with greater expressiveness. An argument for greater expressiveness of interactive models was made by Milner [24], where he stated that the λ-calculus needs to be extended to model interactive systems. Since the λ-calculus models all algorithmic computation, it follows that interactive computation is more expressive. An alternate approach to proving the same result, based on *Persistent Turing Machines*, can be found in [18].

When an interactive system consists of many autonomous (or asynchronous) concurrent components, it cannot in general be simulated by interleaving the behaviors of its subsystems.

> *Distributed interaction is more expressive than sequential interaction, just as sequential interaction is more expressive than algorithmic computation.*

Multi-component systems are capable of richer behaviors than sequential agents. Their behaviors are known as *emergent*, since they emerge as the property of the whole system without being present, in whole or part, in any of its components. The existence of emergent behaviors was demonstrated by Simon [34]; while he discussed *complex systems* in general, interactive computing systems are a special class of such systems.

Infinity of resources. The Turing Machine model can be extended by removing any *a priori* bounds on its resources, possibly resulting in:

- an infinite initial configuration,
- an infinite architecture,
- infinite time,
- an infinite alphabet.

The impracticality of possessing infinite resources should not be an obstacle here. Just as Turing allowed infinite length of tape in Turing Machines, and cellular automata are allowed to contain infinitely many cells, we can allow an infinite number of tapes or states in our models of computation. And just as the infinite length of the Turing Machine tape allows for more expressiveness than bounded-tape models (section 2.2), these extensions increase expressiveness yet further.

Below, we discuss the four different types of *extension by infinity.*

- Persistence of memory between computation is represented by cellular automata, Persistent Turing Machines [17,18], and \$-calculus [10]. When the Turing Machine preserves some information from one computation to the next, we can obtain an unbounded growth of its initial configuration, and we need to model it with an *infinite initial configuration.*
- When modeling massively parallel scalable computers or the Internet, we do not put restrictions on the number of computing elements. Allowing infinitely many computing elements (infinity of architecture) can be modeled by an *infinite number of Turing Machine tapes*, or an *infinite number of read/write heads*, resulting in an unbounded parallelism. The approach is represented by cellular automata [45], discrete neural networks with an infinite number of cells, random automata networks [15], π-calculus [23], and \$-calculus [10]. Just as the large memories of digital computers provide a practical approximation to Turing Machines' infinite tapes, so does system scalability, such as scalable massively parallel computers or dynamic networks of autonomous agents, provide a practical approximation to the infinite architecture of super-Turing computers.
- Any system that is not expected to halt on its own needs to be modeled by allowing *infinite time.* This applies to many interactive systems such as operating systems, servers on the Internet, software agents or viruses, or evolutionary programs.
- Allowing *infinite precision* is represented by analog computers, neural networks and hybrid automata [33]. Analog computers or real-value neural networks can be interpreted as operating on uncountable alphabets – each real number corresponding to one unique symbol of the alphabet. Alternately, real numbers can be simulated by infinite strings over finite discrete alphabets, but then we trade one type of infinity for another. For the same practical reasons why evolutionary programs are terminated after a finite number of generations, current digital computers require truncating all real numbers to finite precision.

Evolution of the system. *Extension by evolution* allowing the computing device to adapt over time. The Turing Machine stops being static but continuously evolves, so it is able to solve new types of problems. Turing's unorganized machine learning [41] can be viewed as an example of this. He proposed strategies, now known as *genetic algorithms*, to evolve the connections between the neurons within the u-machine.

In general, evolution can be done by upgrade of either hardware or software, by self-adaptive, learning programs, or evolvable, self-reproductive hardware. Evolution may happen in continuous or discrete steps, leading possibly to capabilities previously not present. In particular, an ordinary Turing Machine can evolve to one with an oracle, or to a persistent TM that does not reinitialize its tape before computations, or one that replicates its tapes indefinitely. The possibilities of evolution (the types of variation operators) are endless.

Evolution can be controlled by interaction with the environment, or by some performance measure, such as its *fitness*, or *utility*, or *cost*. The evolution principle is used by *site and internet machines* [44], *$-calculus* [10], and Turing's *u-machines* [41].

Discussion. The three principles we have identified and discussed above are consistent with the work of other researchers in this area. For example, in their search for more expressive models of computation, van Leeuwen and Wiedermann [44] stressed:

- interaction of machines;
- infinity of operators;
- non-uniformity of programs (upgrade of computer hardware and system software), incorporated here as part of the evolution principle.

Each of the three extensions is sufficient to obtain models more expressive than Turing Machines. However, the three approaches are not disjoint; it is impossible to have evolution without infinity, or to benefit from infinity without interaction. It is not clear whether our list of possible Turing Machine extensions is complete. At this point, we are rather interested that such extensions are reasonable, and that they cover all models discussed in next section.

3.4 Examples of super-Turing computation

We now present three examples of super-Turing computation requiring new models of computation going beyond Turing Machines:

- dynamic interaction of clients and servers on the Internet;
- mobile robotics;
- infinite adaptation of evolutionary computation.

Distributed Client-Server Computation. The Internet connects many separate computer networks. The *client/server model* is a typical paradigm used by computer networks. In this model, servers provide services to multiple clients (e.g., in the form of web browsers); the clients query the server simultaneously, unaware of the presence of other clients. With the Internet, each client can gain access not just to its local server, but to any server on the Internet, and interact with it as with its local server.

The resulting concurrent interaction of multiple clients and servers, with a dynamic configuration of communication links and nodes, cannot be described as a Turing Machine computation, which must be sequential, static, and with all input predefined.

It can be argued that everything is a matter of providing the proper initial description of the *world* on the infinite tape of the Turing Machine. While there is no bound on the length of the input string, the input for any given problem instance must be finite and predefined ahead of computation. By contrast, the potential interaction streams for dynamic systems may not only be infinite, but even non-enumerable [46,47]. The input values in this infinite dynamic stream depend on the current state of a potentially ever-changeable and uncomputable world, which in turn depends on the earlier output values of the interactive system. Thus Turing Machines can only approximate interaction on the Internet, but cannot be used as its precise model.

Mobile robotics. Mobile robots can be viewed as computers augmented with sensors and actuators to perceive their environment and to physically act upon it. Robots interact with environments which are often more complex than robots themselves; in fact, the environments can be noncomputable, e.g. when they include human actors.

The original approach of artificial intelligence, now known as GOFAI (good old fashioned artificial intelligence), was to implement the robot algorithmically. *Deliberative symbolic robotics* precomputed all the robot's actions before any were carried out, encoding the robot's environment as predefined input. It is not surprising that GOFAI failed miserably with mobile robots.

If the environment can be noncomputable, it can neither be predefined, nor computed using a Turing Machine. This is not just an issue of the enormous computational complexity required to build good models of reality, but of the impossibility of doing it in finite time using finite encodings, as in algorithmic computation. Heeding Brooks' persuasive arguments against the algorithmic "toy world" approach for building reactive robots [3], AI has made a paradigm shift in the last decade towards the interactive approach for building intelligent agents.

Evolutionary computation. Turing Machines have problems with capturing evolution, both natural and that simulated on computers, because

Evolutionary computation can be understood as a probabilistic beam search, looking for the solution with a globally optimal value of the *fitness function*. The fitness function represents the quality of the solution, and is obtained by the process of interaction of the solution with its environment. Populations of solutions and evolutionary algorithms are changed in each *generation*, and the probabilistic search for a solution is, in a general, an infinite process.

When the best solutions are preserved from generation to generation (the so-called *elitist strategy*), evolutionary computation is guaranteed to converge to its goal in the infinity; otherwise, there is no guarantee that the goal will be reached. Usually, evolutionary computation is terminated after a finite number of generations, producing approximate solutions. The halting of genetic algorithms is enforced by the necessity to fit them to the Turing Machine model of computation, with its finiteness restrictions.

An infinite process cannot be properly captured using finitary means (algorithms and Turing Machines). The lack of finitary termination applies also to reactive systems (operating systems and servers). When viewed as a computing system, the Internet also escapes finitary description; its size, as measured by the number of nodes, in principle has no bounds and can "outgrow" any finite description that we may devise for it. This is analogous to the fact that Turing Machines need an infinite tape for their expressiveness (section 2.2) even though their tape contents is always finite.

4 Models of super-Turing computation

In this section we discuss three formal models of super-Turing computation: *Persistent Turing Machines*, the *π-calculus* and the *\$-calculus*. At the end of this section we present an overview of some other super-Turing models of computation, for a more complete perspective.

Regarding the feasibility of implementing super-Turing models, we note that the same criticism applies to Turing Machines. In 1936, when Turing Machines were invented, no one knew how to build a computer. Even now, when digital processors are ubiquitous, no one has implemented a Turing Machine, since that would require infinite memory.

While technically not implementable (due to the infinite size of their tape), Turing Machines serve as a very useful theoretical tool for understanding the algorithmic computation of digital computers. Similarly, super-Turing models are useful for providing a theoretical understanding of super-Turing computation that involves elements of *interaction*, *infinity*, and *evolution* (section 3.3).

4.1 Persistent Turing Machines

Persistent Turing Machines (PTMs) are a model of *sequential interactive computation* obtained by a minimal extension of the Turing Machine model [17,18]. While the "hardware" (syntax) of the PTM is not new, the way we interpret its computation (semantics) is. A PTM is a 3-tape Turing Machine, with an *input*, an *output*, and a *work tape*. It continuously interacts with its environment during the computation, processing a stream of *inputs* that are dynamically generated by the environment, and producing the corresponding stream of *output tokens*. Upon receiving an input token from its environment, the

PTM computes for a while and then outputs the result to the environment, and this process is repeated forever.

A PTM is *persistent* in the sense that its work-tape contents (its "memory") is maintained from one computation to the next. Hence, initial configuration of the PTM is not identical for each input token, as it would be for a Turing Machine. Since the value of the output depends both on the input and the memory, PTMs exhibit history-dependent behavior that is typical of interactive computation, but impossible for algorithmic computation (section 3).

The following is a simple yet practical example of a PTM, an *answering machine* (AM). AM's memory is the tape of the answering machine; it is unbounded. AM's computational steps are expressed by a Turing computable function mapping the pair (input token, current memory) to the pair (output token, new memory):

Example 1. An *answering machine AM* is a PTM whose work tape contains a sequence of recorded messages and whose operations are `record`, `playback`, and `erase`. The Turing-computable function for A is:

$f_A(\texttt{record}\ \mathrm{Y}, \mathrm{X}) = (\texttt{ok}, \mathrm{XY})$; $f_A(\texttt{playback}, \mathrm{X}) = (\mathrm{X}, \mathrm{X})$;
$f_A(\texttt{erase}, \mathrm{X}) = (\texttt{done}, \epsilon)$.

To illustrate AM's ability to express history dependent behavior, consider the input stream

```
(record A,playback,record B,playback,...);
```

the corresponding output stream is

```
(ok,A,ok,AB,...).
```

The same input token (*playback*) generated different output tokens (A and AB) when called at different times, because of the different operations that preceded it.

The notion of *equivalence* for PTMs is *observational*, based only on the observable aspects of its behavior: inputs and outputs; memory is not considered observable. As the example above shows, the whole input stream needs to be observed, rather than just the current token as for a Turing Machine. The resulting notion of equivalence is analogous to that of *bisimulation* for *concurrent processes.*

Alternate notions of equivalence can be defined, based on *finite trace observations* rather than *infinite stream observations.* In fact, there is an *infinite hierarchy* of successively finer equivalence classes for PTMs, when observed over finite stream prefixes. This is in contrast to Turing Machines, which admit only one notion of equivalence, corresponding to the equivalence of the function being computed.

The answering machine example illustrated how PTM constitutes a natural model for sequential objects. Mobile intelligent agents are also naturally modeled by PTMs. Dynamically bounded input and output streams allow us to express uncertainty of both sensors and actuators, while the contents of the persistent worktape "evolves" to reflect the agent's changing *state* as it adapts to its environment.

While distributed interaction is not captured by PTMs, one can imagine how to define systems of PTMs that model distributed interaction. PTMs would be equipped with additional *communication tapes.* To allow dynamic reconfiguration of PTM systems, new primitives for creating, sharing, and modifying communication tapes are needed, but the underlying "hardware" of the system would still be based on Turing Machines.

4.2 The π-calculus

Milner's π-calculus [23,24,29] is a mathematical model of concurrent processes, which are *mobile* in the sense that their interconnections dynamically change during the computation. π-calculus can be considered a dynamic extension of CCS (Calculus of Communicating Systems), Milner's earlier model which was not mobile [22].

Similar to PTMs (section 4.1), the π-calculus is built around the central notion of *interaction.* π-calculus rests upon the primitive notion of interaction via message passing, just as Turing Machines rest upon the notion of reading and writing a storage medium, and just as recursive functions and the λ-calculus rest upon mathematical functions. It is a model of the changing connectivity of *interactive systems*, where *handshaking* allows for synchronous message passing among asynchronous processes.

The ability to directly represent mobility, allowing processes to reconfigure their interconnections while they execute, makes it easy to model networked systems of mobile agents, or any other system where communication links and other resources are allocated dynamically.

π-calculus subsumes Church λ-calculus; by the Church-Turing thesis, it follows that π-calculus is at least as expressive as Turing Machines. It actually is more expressive, extending Turing Machines along all three dimensions (section 3.3): *interaction* (between processes), *infinity* (of time for process computation), and *evolution* (of the communication topology). This is evidenced by the existence of computable non-algorithmic problems such as *driving home from work*, as well as by Milner's own assertion that "a theory of concurrency and interaction requires a new conceptual framework" and cannot be expressed by Turing Machines [24]. Unfortunately, there is yet no definite metrics for measuring the expressiveness of the π-calculus [29].

π-calculus is closely related to Hoare's CSP [19]. Both have been implemented in practice. The OCCAM programming language is based on CSP; the PICT programming language implements an asynchronous version of the

π-calculus [28]. However, π-calculus has yet to gain popular acceptance or commercial success.

4.3 The $-calculus

$-calculus (pronounced "cost calculus") is a mathematical model of *interactive process-based problem solving*, capturing both the final outcome of problem solving as well as the interactive incremental process of finding the solution. It was introduced in the 1990's [9–11] as a formalization of resource-bounded computation, also known as *anytime algorithms.*

Anytime algorithms, proposed by Dean, Horvitz, Zilberstein and Russell in the late 1980's [20], are approximate algorithms whose solution quality increases monotonically as more resources (e.g., time or memory) are available.

$-calculus is a process algebra for *bounded rational agents.*

Just as π-calculus was built around a central concept of *interaction*, $-calculus rests upon the primitive notion of *cost.* An example of cost is the notion of *power consumption* for *sensor network querying.* Power is consumed at a different rate for different activities of the sensor, including communication; minimization of overall power consumption drives the design query evaluation and optimization for sensor networks. $-calculus provides support for problem solving via an incremental search for solutions, using cost to direct the search.

$-calculus has the same primitives to model interaction as π-calculus, communication by message passing, and the ability to create and destroy processes and communication links. In addition, its cost mechanism permits the modeling of resource allocation, or quality-of-service optimization. $-calculus is therefore applicable to robotics, software agents, neural nets, and evolutionary computation. Potentially, it could be used for design of cost-based programming languages, cellular evolvable cost-driven hardware, DNA-based computing and molecular biology, electronic commerce, and quantum computing.

$-calculus expresses evolution naturally. Its cost performance measure models the fitness function. Additionally, $-calculus can simultaneously look for the best solution and lowest search cost. That is, it provides direct support for optimizing not only the outcome of problem solving, but the problem solving methods used to obtain the results.

It is expected that the acceptance of super-Turing computation will result in new classes of programming languages. *Cost languages*, based on the $-calculus model of computation, introduce a new programming paradigm. They are inherently parallel, highly portable, flexible, dynamic, robust, modular, with a 3D graphical interface [9]. The flexible and automatic cost optimization mechanism distinguishes them from other agent related languages. For example, costs can be modeled with probabilities or fuzzy set membership to represent uncertainty.

4.4 Other Super-Turing Models

In this section, we provide a survey of other super-Turing models of computation.

C-machines, o-machines and u-machines. *Choice machines*, *oracle machines*, and *unorganized machines* are three super-Turing models of computation defined by Turing [37,38,41] (section 2). Choice machines and oracles machines derive their expressiveness from the *interaction* principle; unorganized machines, assuming an infinite supply of neurons, derive their expressiveness from the *infinity* and *evolution* principles.

Cellular automata. Cellular automata [45,4] (CAs) were introduced by John von Neumann in search of models for self-reproduction, universal computability, and universal constructibility. They consist of an infinite number of *cells*, where each cell is a finite state machine. CAs are known to be *computation universal*, referring to their ability to implement any algorithm. While CAs can simulate any Turing Machine, the reverse is not true; there exist problems solvable by CAs, but not by TMs. For example, the recognition that an infinite sequence of 9's after a decimal point, i.e., 0.9999..., is another representation of 1, or the problem of the universal constructability requiring the ability to build a clone copy of the TM by the TM itself, cannot be solved by TMs. However CAs can do that [15,45].

The greater expressive power of CAs is due to an unbounded number of cells – the *infinity* principle. Note that whereas the unbounded TM tape only holds a finite string at any one time, CAs encode an infinite string. CAs are also *construction universal*, which refers to their ability to construct arbitrary automata, including themselves (i.e., self-reproduction). Construction universality is one of the cornerstones for research in *artificial life* and *robotics*.

Site and Internet Machines. Van Leeuwen [44] introduced Site and Internet machines to capture the properties of computation that are not modeled directly by Turing Machines:

interaction of machines (our *interaction*),
non-uniformity of programs (our *evolution*),
and infinity of operations (our *infinity*).

Site machines are generally interactive Turing Machines with *advice*, which is a limited (but nevertheless more powerful than TM) version of Turing's oracle. They are equipped with several input and output ports via which they communicate by sending and receiving messages. In particular, a site machine may exchange messages with the advice oracle, to obtain uncomputable advice from it.

An *Internet machine*, as its name implies, models the Internet; it consists of a finite but time-varying set of Site machines that work synchronously and communicate by exchanging messages. Each machine in the set is identified by its address, and the size of the set can grow at most polynomially with time.

Van Leeuwen proves that his *Internet machine* is not more expressive than a single *Site machine*; however, this proof sacrifices the dynamic nature of messages and assumes they can be precomputed. However, both *Site* and *Internet machines* are more powerful than TMs. This is generally due to the power of advice, which represents the uncomputable nature of the site machine's environment.

Analog neural networks. Neural Networks (NNs) consist of computational cells (called *nodes* or *neurons*), much like cellular automata. However, the underlying network is no longer homogeneous but an arbitrary digraph; the transition functions compute weighted sums of inputs from neighboring cells. Depending on whether they operate over discrete or real values, neural networks can be *discrete* or *analog.*

Despite having finitely many cells, analog NNs neural networks can compute nonrecursive functions. In particular, they can compute the *analog shift map*, known to be non-recursive [33]. In fact, neural networks can be a standard model for super-Turing analog computing, analogous to the role of the Turing Machine in the Church-Turing thesis:

> *No possible abstract analog device can have more computational capabilities (up to polynomial time) than first-order (i.e., the net neuron function is a weighted sum of its inputs) recurrent neural networks.* [33].

The extra power of analog NNs stems from their real-valued weights, allowing the neurons to take continuous values in their activation functions. The implementation of such a neural network can be viewed as an idealized chaotic physical system.

Evolutionary Turing Machines. By an *Evolutionary Turing Machine* (ETM) [12], we mean a (possibly infinite) series of Turing Machines, where the outcome tape from one generation forms the input tape to the next generation. In this sense ETM resembles a Persistent Turing Machine. The tape keeps both a population of solutions, and the description of an evolutionary algorithm. Both population of solutions and evolutionary algorithms can evolve from generation to generation. The goal (or halting) state of ETM is represented by the optimum of the fitness performance measure. The higher expressiveness of ETM is obtained either evolving infinite populations, or applying an infinite number of generations (the infinity principle), or applying variation (crossover/mutation) operators producing nonrecursive solutions (the evolution principle).

Accelerating Universal Turing Machines. Another super-Turing model is the *Accelerating Universal Turing Machine* (AUTMs) [7], where programs are executed at an ever-accelerating rate, permitting infinite programs to finish in finite time. AUTMs require a maximum of two time units to execute any possible program; for example, the first operation takes 1 time unit, the second 0.5 time unit, the third 0.25, and so on.

5 Towards a new kind of computer science

In this section, we will speculate on the effect of the paradigm shift to Super-Turing computation on several areas of computer science: artificial intelligence, programming languages, and computer architecture.

5.1 Super-Turing intelligence: interactive extensions of the Turing Test

When Turing proposed his famous Turing Test, he raised the question: *Can machines act intelligently?*, which he then answered in the affirmative (section 2.7). Not everyone shared Turing's optimism. Skeptics such as Penrose [27] argued that Turing Machines cannot simulate the extensional behavior of humans or physical systems, believing that computers' behavior is essentially too weak to model intelligence.

Replacing Turing Machines with interactive systems, either *sequential* or *distributed* (section 3.3) in the Turing Test allows us to model stronger extensional behavior. The *interactive Turing Test* preserves Turing's behaviorist assumption that thinking is specifiable by behavior, but extends models of questioning and responding to be interactive [46,48]. Analysis of interactive question answering yields behaviorist models of "thinking" that are qualitatively stronger than the traditional, algorithmic, Turing Test model.

algorithmic Turing Test: measures ability to answer a set of unrelated *questions* from a predetermined script;
sequential Turing Test: measures ability to carry out a *dialogue* involving follow-up questions, where the answers must show adaptive history-dependent thinking;
distributed Turing Test: measures ability to carry out *projects* involving coordination and collaboration for multiple autonomous communication streams.

Algorithmic, sequential, and multi-agent thinking are progressively more powerful forms of behavioral approximation to human thinking, defined by progressively more stringent empirical tests of behavior. While we agree with Penrose's arguments [27] against the algorithmic Turing Test, we believe they no longer hold for the stronger interactive forms of the test.

The interactive Turing test extends machines without changing the form of the Turing test. Turing would certainly have accepted the interactive versions of the Turing Test as legitimate, and would have approved of the behaviorist notions of sequential and distributed thinking as conforming to the spirit of his notion of machine intelligence.

We also consider *infinite and evolutionary* versions of the Turing Test. The first will allow the testing to continue for an indefinite period of time. Like elections for the presidency in a country without term limits, the system needs to continue proving itself over and over in order to avoid failing the test. An evolutionary version is even stronger; the rules for intelligence evolve as the test progresses, and the system is expected to adapt to the new rules.

Russell and Norvig [31] propose another extension: a *total Turing Test* where the the computer has the ability to move about, to perceive its surroundings, and to manipulate objects. This would free the tester from the need to interact with the system via an artificial interface, as well as allow her to test the system's ability to act within and on its environment.

5.2 Super-Turing architectures

The architecture for super-Turing computers has the following three requirements, reflecting the three principles of super-Turing computation (section 3.3):

- **(interaction principle)** Highly interactive both with the environment and other computers.
- **(infinity principle)** Highly scalable – allowing use of more and more time, memory, and other resources in the computation. There should be no limit on that, i.e., computers will be closer and closer to satisfy in the limit the infinity principle.
- **(evolution principle)** Highly adaptable allowing modification of both hardware and software. Evolvable hardware and software may lead in a more natural way to computers that learn rather than be programmed.

Today's computers are still based on the 50-year-old "von Neumann architecture", which was inspired by the Universal Turing Machine model. There are cosmetic changes, such as multi-level caches and pipelining, but under the hood it remains the same old model. While today's systems are somewhat interactive, they fall far short in each of these three categories.

So far, none of the attempts to build a *supercomputer* based on non-von Neumann architecture was commercially successful. The *Fifth Generation Project* in Japan, Europe, and USA produced many ambitious non-von Neumann architecture designs, from which very few (if any) survived. *Reduction* and *dataflow* computers, attempting to eliminate the so-called *von Neumann bottleneck* and increase the speed of computers, turned out to be inefficient. *Cellular computers* had their peak with the massively parallel architectures of *Connection Machines*. Their creator, the Thinking Machine Corporation, is no longer in the business of building parallel processors. Cray Computer Corporation, another manufacturer of massively parallel architectures, has likewise declared bankruptcy. Intel and DEC have also stopped manufacturing supercomputers.

We conjecture that a single computer based on a parallel architecture is the wrong approach altogether. The emerging field of *network computing*, where many autonomous small processors (or *sensors*) form a self-configured network that acts as a single distributed computing entity, has the promise of delivering what earlier supercomputers could not. This is consistent with Turing's design of the ACE [39,40], where Turing advocated putting the whole complexity into software keeping the hardware as simple as possible.

An evolvable and self-reconfigurable network is not a single machine, but it constitutes a single distributed computing system, which can be mobile, embedded, or environment-aware. These systems are the basis for the *ubiquitous* and *pervasive* computing applications which are expected to be the "killer apps" of the next generation. They get us closer to von Neumann's Theory of Self-Reproducing Automata [45], Turing's morphogenesis theory [43], or the total Turing Test (section 2.7).

5.3 Programming languages for super-Turing computers

We can identify four paradigms for programming languages, in the order of their appearance:

- procedural
- functional
- declarative (logic, constraints)
- object-oriented (sequential, asynchronous)

This classification applies to high-level languages. Very rarely is it applicable to assembly languages and machine languages, which tend to be uniformly procedural.

The object-oriented approach allows the "encapsulation" of other programming paradigms, creating hybrid o-o-procedural, o-o-functional, and o-o-declarative languages. It also provides nice support for the development of graphical user interfaces, and it is claimed to be an appropriate implementation tool for interactive parallel programming. Have we found in object-oriented programming the universal paradigm to program super-Turing computers?

How should the programming languages for super-Turing computers look? The three principles of super-Turing computation, *interaction*, *infinity*, and *evolution* (section 3.3), serve as beacons in our search for an answer:

- Programming languages for super-Turing computing should be highly dynamic, to express learning and programming in rapidly changing environments, and to specify highly interactive applications.

- Programs should adapt automatically to the environments and new goals.
- These languages should provide support for solving problems expressed in a declarative fashion.
- They should have built-in optimization engines for hard search problems that may arise during computation, much more powerful than the depth-first search strategy used in Prolog.
- They should be able to solve *in the limit* currently undecidable problems; that is, to obtain approximate solutions with arbitrarily high precision, or to obtain correct solutions probablistically with arbitrarily low chance of error.

Most likely, object-orientation is here to stay, but not in its current form. When mobile embedded networked sensors are modeled as asynchronous concurrent objects, it is clear that their programming will require new primitives so interaction (communication) and mobility can be handled explicitly. Furthermore, we conjecture that constraints will play a greater role in the programming languages of the future, to enable declarative specification of desired system behavior. To translate these specifications into executable code, the languages will also have built-in dynamic search strategies and approximate solvers for undecidable problems, as discussed above. This will require new computing technologies, such as analog, optical, biological, or quantum technologies, which will be part of super-Turing architectures.

6 Rethinking the Theory of Computation

While super-Turing systems model new forms of computation, they may allow us to find new solutions for old problems. In this section, we look at some aspects of theoretical computer science that will be affected by the paradigm shift to super-Turing computing. While these ideas sound rather futuristic, they would have been welcomed by Turing, who always looked beyond the horizon of practical feasibility of the moment.

6.1 Computing the undecidable

The *halting problem* was the first problem identified by Turing as unsolvable [37]. Thus it is typical for super-Turing models to demonstrate their higher expressiveness by the solution of the halting problem. The proofs use either interaction, infinity or evolution principles.

The first such proof was given by Alan Turing himself, who demonstrated that his o-machine can solve (nonalgorithmically) the halting problem, by interacting with an oracle. Later, Garzon showed how to solve the halting problem by an infinite number of discrete neurons [15]. In [11], three ways are presented for solving the halting problem in *\$-calculus* – using either interaction, or infinity, or evolution principles.

Hava Siegelmann departed from the halting problem, and showed that real value neural networks can solve a non-recursive analog shift map using an infinity principle [33].

Note that none of these solutions of the Halting Problem (H) are algorithmic. In the proofs, either some steps of the solutions do not have well defined (and implementable) meaning – this is when we refer to the help or oracles – or we use infinite resources (an infinite number of steps requiring perhaps an infinite initialization time).

Also note that, while the undecidability of H led us to prove the undecidability of many other problems, its decidability does *not* have similarly wide implications. Given that H is undecidable, the proof that some other problem P is undecidable relies on an argument *by contradiction*: if P were decidable, then H would be too. However, if the original problem H turns out to be solvable (using more powerful means), it does not imply the solvability of the other problem.

6.2 Is the Church-Rosser theorem still valid?

According to the Church-Turing thesis, Turing Machines are equivalent to recursive functions. The first *Church-Rosser theorem* [6] states that:

> Given a function, no matter which evaluation order of the function parameters is chosen, the result will be the same (up to renaming of bound variables) as long as the computation terminates.

The possible interpretation of this theorem can be the following: if the sequence does not matter, then the evaluations can be done in parallel without worrying about the order in which they finish because the final outcome will be the same. The above is typically interpreted that sequential programming is equivalent to parallel programming because both lead, according to the Church-Rosser theorem, to the same results.

However, we must realize that the Church-Rosser theorem is valid for so-called "pure" functional computation without side effects. By contrast, both assignments statements from procedural languages as well as interaction by message-passing from object-oriented languages entail side effects. For languages that allow side effects, it has been assumed that an equivalent program without side effects can always be found, to ensure the applicability of the Church-Rosser theorem.

While this assumption has long been seen as overly optimistic, proving it false would be equivalent to a rejection of the Church-Turing thesis. We believe it to be false. This follows from the fact that interaction is more expressive than algorithms, and that distributed interaction is more expressive than sequential interaction.

> *The Church-Rosser theorem, while valid for algorithmic computation, is not valid for super-Turing computing.*

One of the most important principles of super-Turing computation is interaction, which is not side-effect-free. In fact, the side effects are crucial for the power of interaction, such as its ability to express *emergent behavior*. For example, many scientists have pointed out that the complexity of the behavior of an ant colony (which is a distributed interactive system) is caused by side effects of their interaction rather than by the complexity of ants themselves.

6.3 Rewriting complexity theory: Is the $P = NP$ question still relevant?

The class P consists of all those algorithmic problems solved by some *deterministic* TM whose running time is polynomial in the size of the input. The class NP can be defined in two ways – either as an analogue of P, but with a *nondeterministic* TM, or as those algorithmic problems whose solutions, if given to us by others, can be *verified* by some deterministic TM in polynomial time.

The question of whether $P = NP$ has been occupying researchers since these two sets were first defined, having become the the greatest unsolved question of computer science. While it is simple to show that P is a subset of NP, the other direction remains a mystery. The problems in P are considered to be easy and tractable; those outside of P are not. Are there problems in NP which are *not* in P? Many NP problems have been identified that appear *unlikely* to be in P, but without a proof. Among them are the *Traveling Salesman Problem* (is there a tour of all the nodes in a graph with total edge weight $\leq k$?), SAT (does a Boolean expression have a satisfying assignment of its variables?), and CLIQUE (does a graph have a set of k nodes with edges between every pair?).

Donald Knuth believes that $P = NP$, and that we will eventually find a *nonconstructive* proof that the time complexity of all NP problems is polynomial, but we will not know the degree or the coefficients of this polynomial. We conjecture that the $P = NP$ question is inherently undecidable, that it can be neither proved nor disproved. However, the great majority of computer scientists believe that $P \neq NP$ [16]. Most, including Jeff Ullman, believe that it will take another hundred years to prove it, because we have not yet invented the right techniques. But some, including Richard Karp, think that the problem will be solved earlier, by a young researcher with non-traditional methods, unencumbered by the conventional wisdom about how to attack the problem – something in the spirit of Alan Turing, who never followed conventional paths.

We expect that the $P = NP$ question will lose its significance in the context of super-Turing computation. Being able to interact with the real world during the computation, or to access infinite resources, or to evolve, will reframe computational problems in a non-algorithmic way that shifts the focus away from the $P = NP$ question. So far, the super-Turing research has concentrated on undecidable problems, but the effective means of solving traditionally intractable problems should and will be their equally important objective.

7 Conclusions

Super-Turing models extend Turing Machines to permit interactive input from the environment during computation, thereby modeling problems that cannot be modeled by Turing Machines - like driving home from work, flying airplanes, artificial intelligence, and human thinking. Driving home requires knowledge of the environment not expressible by TMs, flying planes requires environmental knowledge of temperature and other airplanes, artificial intelligence requires knowledge on intelligent tasks, and human thought requires knowledge of human environments during thought. Each of these topics cannot be handled by TMs but can be handled by super-Turing models like Interactive Machines or process algebras like π-calculus.

The adoption of TMs as a model of universal problem solving has raised many computational problems that can be solved by extending TMs to interactive models. AI has for many years been viewed as unsolvable because TMs could not handle certain problems that can be handled by interactive models. The Turing test, proposed by Turing as a basis for human thought, has been criticized by Searle, Penrose and others as being intractable to question answering by TMs but becomes tractable if question answering is extended to interactive systems (section 5.1). The Japanese Fifth-generation computing paradigm was found to be intractable because logic programming could not be expressed by TMs, but could in principle be handled by an extended form of interactive logic programming. Algorithms express enumerable problems that can be constructed, while interaction expresses observable problems that may be nonenumerable.

The transition from enumerable construction to nonenumerable observation requires a mathematical change in models of problem solving that eliminates traditional models based on induction and can be viewed as an elimination of traditional mathematics as a model of computation. Interactive models are not mathematical in the traditional sense, but an extended model of mathematics not yet specified could in principle provide an acceptable model.

Super-Turing models require a change in models of philosophy as well as of mathematics. Philosophers have widely debated the relation between rationalist models of the mind and empiricist models of experimental behavior. Aristotle's model of logic and mathematics is rationalist, while Plato's model of the cave is empiricist, requiring knowledge of the environment for problem solving. Descartes "cogito ergo sum" is rationalist while Hume's model is empiricist, and was accepted by Kant in his book "Critique of Pure Reason" as an empiricist principle of reasoning as a form of problem solving. Hilbert and

Russell's models of mathematics are rationalist, while Gödel and Turing's proof of unsolvability of the Entscheidungsproblem is empiricist. Mathematical acceptance of Turing Machines as a universal model for all computation can be viewed as a rationalist transformation of Turing's empiricist model. Interaction is an empiricist model of computation consistent with Gödel and Turing's original models and with the need of future computer models. The importance of interaction as a model of computation corresponds to the importance of empiricism over rationalism as a principle of philosophy.

Robin Milner has asserted [25] that Turing would have been excited by the direction in which computing has evolved; we agree. In particular, we believe that Alan Turing would approve of super-Turing models, and he would be the first to argue that they, rather than the Turing Machine, represent the future of computing.

We expect super-Turing computation to become a central paradigm of computer science. However, we cannot claim that super-Turing models, as described in this chapter, are definitive and complete. Most likely, they will be superseded in the future by models that are even better and more complete for problem solving, in the never ending quest for a better description of reality.

References

1. Bäck T., Fogel D.B., Michalewicz Z. (eds.), *Handbook of Evolutionary Computation*, Oxford University Press, 1997.
2. "Algorithm." *Encyclopedia Britannica 2003* Encyclopedia Britannica Premium Service. `<http://www.britannica.com/eb/article?eu=5785>`.
3. Brooks R.A., Elephants Don't Play Chess, in P.Maes (ed.), *Designing Autonomous Agents: Theory and Practice from Biology to Engineering and Back*, The MIT Press, 1994, pp. 3-15.
4. Burks A., *Essays on Cellular Automata*, Univ. of Illinois Press, 1970.
5. Church A., An Unsolvable Problem of Elementary Number Theory, *American Journal of Mathematics* 58, 1936, pp. 345-363.
6. Church A., Rosser J.B., Some properties of conversion, *Trans. AMS* 39 (1936), pp. 472-482.
7. Copeland B.J., Super-Turing Machines, *Complexity*, 4(1), 1998, pp. 30-32.
8. Martin Davis, Why G"odel Didn't Have Church's Thesis, *Information & Control* 54, 1982, pp. 3-24.
9. Eberbach E., Brooks R., Phoha S., Flexible Optimization and Evolution of Underwater Autonomous Agents, LNAI 1711, Springer-Verlag, 1999, pp. 519-527.
10. Eberbach E., $-Calculus Bounded Rationality = Process Algebra + Anytime Algorithms, in: (ed.J.C.Misra) Applicable Mathematics: Its Perspectives and Challenges, Narosa Publishing House, New Delhi, Mumbai, Calcutta, 2001, pp. 213-220.
11. Eberbach E., Is Entscheidungsproblem Solvable? Beyond Undecidability of Turing Machines and Its Consequence for Computer Science and Mathematics, in:

(ed.J.C.Misra) Computational Mathematics, Modeling and Simulation, Narosa Publishing House, New Delhi, Chapter 1, 2003, pp. 1-32.

12. Eberbach E., On Expressiveness of Evolutionary Computation: Is EC Algorithmic?, *Proc. 2002 World Congress on Computational Intelligence* (WCCI), Honolulu, HI, 2002, pp. 564-569.
13. Farley B.G., Clark W.A., Simulation of self-organizing systems by digital computer, *IRE Transactions on Information Theory* 4, 1954, pp. 76-84.
14. Fogel D.B. (ed.), *Evolutionary Computation: The Fossil Record*, IEEE Press, NY, 1998.
15. Garzon M., Models of Massive Parallelism: Analysis of Cellular Automata and Neural Networks, An EATCS series, Springer-Verlag, 1995.
16. Gasarch W., Guest Column: The P=?NP Poll, *SIGACT News*, 33:2, June 2002, pp. 34-47.
17. Goldin D., Persistent Turing Machines as a Model of Interactive Computation, FoIKS'00, Cottbus, Germany, 2000.
18. Dina Goldin, Scott Smolka, Peter Wegner, Turing Machines, Transition Systems, and Interaction *proc. 8th Int'l Workshop on Expressiveness in Concurrency*, Aalborg, Denmark, August 2001
19. Hoare C.A.R., *Communicating Sequential Processes*, Prentice-Hall, 1985.
20. Horvitz E., Zilberstein S. (eds.), Computational Tradeoffs under Bounded Resources, *Artificial Intelligence* 126, 2001, pp. 1-196.
21. McCulloch W., Pitts W, A Logical Calculus of the Ideas Immanent in Nervous Activity, *Bulletin of Mathematical Biophysics* 5, 1943, pp. 115-133.
22. Milner R., A Calculus of Communicating Systems, *Lect. Notes in Computer Science* 94, Springer-Verlag, 1980.
23. Milner R., Parrow J., Walker D., A Calculus of Mobile Processes, I & II, *Information and Computation* 100, 1992, pp. 1-77.
24. Milner R., Elements of Interaction, *Communications of the ACM* 36:1, Jan. 1993, pp. 78-89.
25. Milner R., *Turing, Computing and Communication*, a lecture for the 60'th anniversary of Turing's "Entscheidungsproblem" paper, King's College, Cambridge England, 1997.
26. Minsky M.L., Theory of Neural-Analog Reinforcement Systems and Its Applications to the Brain-Model Problem, Ph.D. Thesis, Princeton University, 1954.
27. Penrose R., *The Emperor's New Mind*, Oxford, 1989.
28. Pierce B., Turner D., Pict: A Programming Language Based on the Pi-Calculus, in: Plotkin G. et al (eds.), *Proof, Language, and Interaction: Essays in Honour of Robin Milner*, The MIT Press, 2000, pp. 455-494.
29. Plotkin G., Stirling C., Tofte M. (eds.), *Proof, Language, and Interaction: Essays in Honour of Robin Milner*, The MIT Press, 2000.
30. Rosenblatt F., The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain, *Psychological Review* 65, 1958, pp. 386-408.
31. Russell S., Norvig P., *Artificial Intelligence: A Modern Approach*, 2nd edition, Prentice-Hall, 2003.
32. Samuel A.L., Some studies in machine learning using the game of checkers, *IBM Journal of Research and Development* 3, 1959, pp. 211-229.
33. Siegelmann H., *Neural Networks and Analog Computation: Beyond the Turing Limit*, Birkhauser, 1999.
34. Simon, H.A. *The Sciences of the Artificial.* MIT Press, 1969.

35. Sipser, M. *Introduction to the Theory of Computation*, PWS Publishing Company, 1997.
36. Teuscher Ch., *Turing's Connectionism: An Investigation of Neural Networks Architectures*, Springer-Verlag, 2001.
37. Turing A., On Computable Numbers, with an Application to the Entscheidungsproblem, *Proc. London Math. Society*, 42:2, 1936, pp. 230-265; A correction, ibid, 43, 1937, pp. 544-546.
38. Turing A., Systems of Logic based on Ordinals, *Proc. London Math. Society*, 45:2, 1939, pp. 161-228.
39. Turing A., The ACE Report, in *A.M. Turing's Ace Report of 1946 and Other Papers*, eds. B.E. Carpenter and R.W. Doran, MIT Press, 1986.
40. Turing A., Lecture to the London Math. Society on 20'th February 1947, in *A.M. Turing's Ace Report of 1946 and Other Papers*, eds. B.E. Carpenter and R.W. Doran, MIT Press, 1986.
41. Turing A., Intelligent Machinery, 1948; in *Collected Works of A.M. Turing: Mechanical Intelligence*, ed. D.C.Ince, Elsevier Science, 1992.
42. Turing A., Computing Machinery and Intelligence, *Mind*, 1950.
43. Turing A., The Chemical Basis of Morphogenesis, *Phil. Trans. of the Royal Society* B237, 1952.
44. Van Leeuwen J., Wiedermann J., The Turing Machine Paradigm in Contemporary Computing, in. B. Enquist and W. Schmidt (eds.) *Mathematics Unlimited - 2001 and Beyond*, LNCS, Springer-Verlag, 2000.
45. Von Neumann J., Theory of Self-Reproducing Automata, (edited and completed by Burks A.W.), Univ. of Illinois Press, 1966.
46. Wegner P., Why Interaction is More Powerful Than Algorithms, *Communications of the ACM* 40:5, May 1997, pp. 81-91.
47. Wegner P., Interactive Foundations of Computing, *Theoretical Computer Science* 192, 1998, pp. 315-351.
48. Wegner P., Goldin D., Interaction as a Framework for Modeling, LNCS 1565, April 1999.
49. Wegner P., Goldin D., Computation Beyond Turing Machines, *Communications of the ACM*, April 2003.